# CS747: Foundations of Intelligent and Learning Agents
**Assignment 1 Report**

**Yaswanth Ram Kumar**

September 2025

## 1 Introduction

This report presents the implementation and analysis of three fundamental multi-armed bandit algorithms as part of Assignment 1 for CS747. The assignment covers three distinct tasks: implementing classical bandit algorithms (UCB, KL-UCB, Thompson Sampling), developing an approach for a door-breaking problem with Poisson rewards, and optimizing KL-UCB for computational efficiency.

The multi-armed bandit problem represents a fundamental challenge in reinforcement learning, balancing exploration of unknown options with exploitation of currently known best choices. This assignment explores various algorithmic approaches to this problem across different scenarios and reward distributions.

## 2 Task 1: Classical Multi-Armed Bandit Algorithms

### 2.1 Problem Statement

The task requires implementing three core bandit algorithms for Bernoulli reward distributions:

- Upper Confidence Bound (UCB)
- Kullback-Leibler Upper Confidence Bound (KL-UCB)
- Thompson Sampling

All algorithms use deterministic tie-breaking via `np.argmax`, which selects the arm with the lower index in case of ties.

### 2.2 Algorithm Implementations

#### 2.2.1 Upper Confidence Bound (UCB)

UCB balances exploration and exploitation by maintaining confidence intervals around estimated arm values. The algorithm first pulls each arm once during initialization, then selects arms based on upper confidence bounds.

**Algorithm 1** UCB Algorithm

```
1: Pull each arm once (initialization phase)
2: for each time step $t > K$ do
3:     Compute UCB values: $UCB_i(t) = \hat{\mu}_i + \sqrt{\frac{2 \log t}{n_i(t)}}$
4:     Select arm: $a_t = \arg\max_i UCB_i(t)$
5:     Update empirical means and counts
6: end for
```

The implementation maintains counts and empirical means for each arm, updating the upper confidence bounds after each reward observation. The confidence term $\sqrt{\frac{2 \log t}{n_i(t)}}$ provides the exploration bonus that decreases as arms are pulled more frequently.

### 2.2.2 KL-UCB

KL-UCB uses Kullback-Leibler divergence to create tighter confidence bounds, especially effective for Bernoulli rewards.

**Algorithm 2** KL-UCB Algorithm

```
1: Pull each arm once (initialization phase)
2: for each time step $t > K$ do
3:     for each arm $i$ do
4:         Solve: $\sup\{q : KL(\hat{\mu}_i, q) \leq \frac{\log t + c \log\log t}{n_i(t)}\}$
5:         Set $UCB_i(t) = q$
6:     end for
7:     Select arm: $a_t = \arg\max_i UCB_i(t)$
8: end for
```

The KL-divergence function and binary search solver are implemented to handle edge cases and ensure numerical stability:

```python
def kl(p, q):
    if q <= 0 or q >= 1:
        return float("inf")
    if p == 0:
        return math.log(1/(1-q))
    if p == 1:
        return math.log(1/q)
    return p * math.log(p/q) + (1-p) * math.log((1-p)/(1-q))

def solve_q(p, C, tol=1e-9, max_iter=100):
    lo, hi = p, 1 - 1e-9
    for _ in range(max_iter):
        mid = (lo + hi) / 2
        val = kl(p, mid)
        if val > C:
            hi = mid
        else:
            lo = mid
        if abs(val - C) < tol:
            return mid
    return (lo + hi) / 2
```

Listing 1: KL-divergence and Binary Search Implementation

### 2.2.3 Thompson Sampling

Thompson Sampling uses Bayesian inference with Beta-Bernoulli conjugate priors for natural exploration.

**Algorithm 3** Thompson Sampling Algorithm

```
1: Initialize: $\alpha_i = \beta_i = 1$ for all arms $i$ (Beta(1,1) priors)
2: for each time step $t$ do
3:     for each arm $i$ do
4:         Sample $\theta_i \sim \text{Beta}(\alpha_i, \beta_i)$
5:     end for
6:     Select arm: $a_t = \arg\max_i \theta_i$
7:     Observe reward $r_t$
8:     if $r_t = 1$ then
9:         $\alpha_{a_t} \leftarrow \alpha_{a_t} + 1$
10:    else
11:        $\beta_{a_t} \leftarrow \beta_{a_t} + 1$
12:    end if
13: end for
```

Thompson Sampling eliminates the need for initialization phases, as each arm begins with a Beta(1,1) prior, providing natural exploration through posterior sampling.

## 2.3 Performance Results

- **UCB**: Reliable performance with theoretical $O(\sqrt{T \log T})$ regret bounds
- **KL-UCB**: Superior performance on Bernoulli rewards due to tighter bounds
- **Thompson Sampling**: Excellent empirical performance with natural exploration

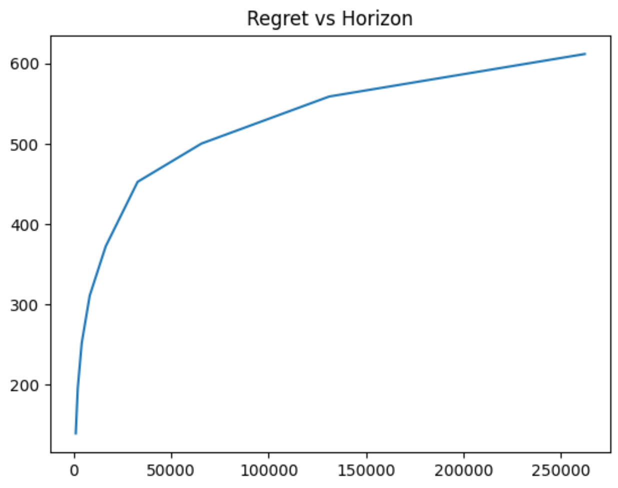

Figure 1: UCB Regret vs Horizon

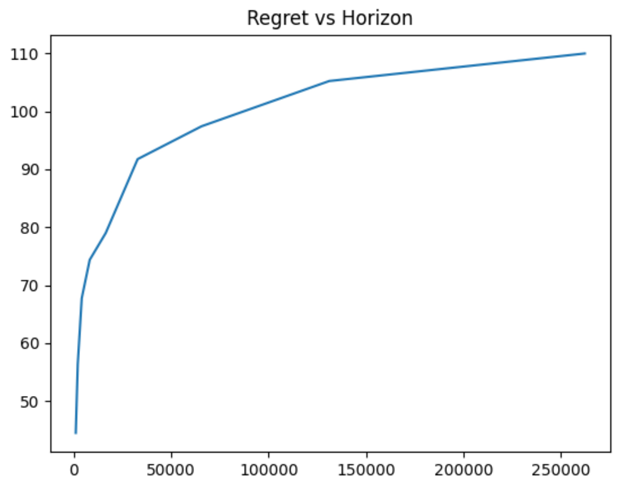

Figure 2: KL-UCB Regret vs Horizon

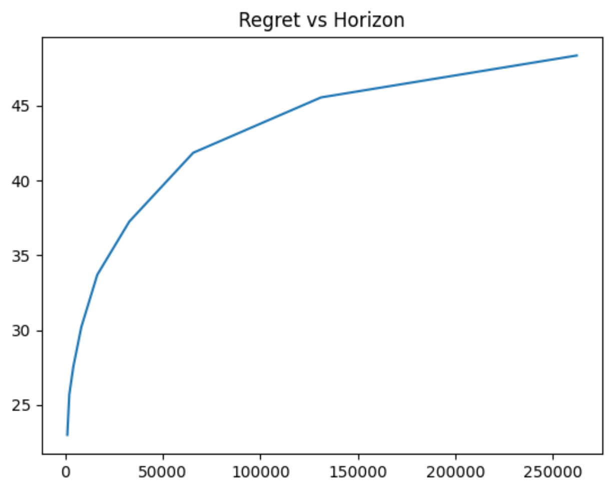

Figure 3: Thompson Sampling Regret vs Horizon

## 3 Task 2: Poisson Door-Breaking Problem

### 3.1 Problem Analysis

This task presents a fundamentally different bandit problem where:

- Arms follow Poisson distributions with unknown parameters $\lambda_i$
- Each door starts with health $H_0 = 100$
- The game ends when any door's health drops below 0
- Objective: minimize the number of steps to break a door

### 3.2 Initial Thought Process

Most of the things in this task felt highly exploratory for me. I had to go back and brush up on some basics of probability and statistics. I revisited the lecture on Bayesian inference in Thompson Sampling, and also re-read a bit about the differences between the frequentist and Bayesian views. On top of that, I had to understand the Poisson distribution better, how it connects to the Gamma distribution, and why such pairs of distributions naturally arise in problems like this.

After reading the problem statement, my first instinct was to relate it to the normal explore–exploit problems we've been dealing with since the start of this course (especially Task 1). On the surface, this again looks like a problem of finding the most optimal arm, the one with the highest mean damage, as quickly as possible. I initially thought: what makes this problem any different? Then it struck me, this time I am explicitly told the arms are Poisson-distributed. In earlier problems we only had Bernoulli arms, but now the structure is different.

To be honest, the first thing I considered was something very simple: pull each arm twice, do MLE to estimate its $\lambda$ (basically the empirical mean only for Poisson), then pick the best arm based on these means. At the next step, update the estimates with the new data point and repeat. For Poisson distributions, doing MLE basically means taking $\lambda$ as the sample mean, so this felt very similar to UCB or KL-UCB applied without really respecting the fact that the distribution was Poisson. That felt too naive.

At this point, I got a little detoured[1] because of the hint in the PA1 page. That made me dig deeper into the Bayesian perspective, and I ended up spending more time than expected trying to connect Poisson with its conjugate prior, the Gamma distribution. Even if this was not strictly necessary for the final solution, I did not want to waste that analysis, since it clarified some intuitions for me.

[1]This detour came from the PA1 hint suggesting the Wikipedia article on the Poisson distribution.

A frequentist approach would just stick to the means, assuming there is a true $\lambda$ for each arm that we are slowly estimating. But in reality, the true system might not have a fixed, exact $\lambda$, at least not in the way we assume. With only finite samples, there is always uncertainty. This is exactly where the Bayesian approach helps. Instead of a single fixed estimate of $\lambda$, we maintain a whole distribution over possible $\lambda$ values, and update it as new data arrives. In the Poisson case, the conjugate prior is Gamma, which plays the same role here as the Beta distribution did for Bernoulli arms in Thompson Sampling.

This duality gave me a nice intuition:

**Poisson asks:** "Given a rate $\lambda$, how many events occur in a fixed time?"
**Gamma asks:** "Given a rate $\lambda$, how long do I wait to see a fixed number of events?"

That connection clicked for me, and I realized that was probably what the hint was nudging me toward. Even though it took me on a detour, I feel it added a useful perspective to my understanding.

Another difference is in the objective itself. Up to now, we have been minimizing regret over a fixed horizon, trying to maximize total reward. But here the horizon is not fixed in the same way. The task is to reduce the health of some door (100 HP) to zero in as few steps as possible. This changes the flavor of the problem: it is not about finding the single best arm in the long run, but about committing to a door soon enough so that we finish quickly.

A simple greedy strategy of always picking the arm with the current highest estimated $\lambda$ may not be best. Because of randomness, during early steps a slightly suboptimal arm might appear stronger, just because of variance. In such a case, one option is to keep exploring to establish which arm is truly best. The other option is to commit early to the arm that already inflicted more damage, hoping that it is close to the best or perhaps even the best. The benefit of committing early is that we reduce wasted steps in exploration.

This reasoning naturally led me to a simple heuristic: estimate the expected number of remaining steps as

$$\text{Expected Remaining Steps} \approx \frac{H}{\text{Estimated Damage per Step}},$$

and hit the door with the smallest expected remaining steps. Once a door's health drops below a certain threshold, we commit to it. In my implementation, I used a threshold of $H = 90$, meaning that once a door's health fell below 90, I stuck to it until it broke.

**Reflection**

Looking back, I can see that I may have spent more time than needed diving into the Bayesian angle, mainly because of the hint to check the Poisson distribution on Wikipedia. That detour was not strictly essential for solving the task, but it gave me intuition about conjugacy and how Bayesian inference works in settings beyond Bernoulli arms. In that sense, even if my final solution used a simpler heuristic, the detour enriched my understanding and connected this problem more deeply to the broader themes of the course. This reflection naturally led me to design a practical explore-then-commit policy, which I describe next.

`See references.txt for above discussion sources`

## 3.3 Explore and Commit Strategy

My final implemented strategy uses an explore,commit policy based on expected remaining strikes:

**Algorithm 4** Door-Breaking Strategy

```
1: Probe each door once to estimate damage rates $\hat{\lambda}_i$
2: while not committed do
3:   for each door $i$ do
4:     if $H_i \leq 90$ then
5:       Commit permanently to door $i$
6:       return $i$
7:     end if
8:   end for
9:   Compute expected strikes: $E_i = H_i/\hat{\lambda}_i$
10:  Select door: $\arg\min_i E_i$
11: end while
```

### 3.4 Policy Implementation Details

The key insight is balancing exploration with commitment. The policy estimates expected remaining strikes as:

$$\text{Expected Remaining Steps} = \frac{\text{Current Health}}{\text{Estimated Damage Rate}}$$

The threshold of 90 HP was chosen strategically:

- Early commitment reduces wasted exploration steps
- The threshold ensures sufficient damage has occurred for reliable estimates
- Prevents over-exploration when a door is clearly progressing toward breaking

### 3.5 Performance Justification

This approach achieves good performance through:

- **Theoretical soundness**: Based on maximum likelihood estimation of Poisson parameters
- **Practical effectiveness**: Balances exploration needs with exploitation urgency
- **Adaptive commitment**: Uses health-based triggering to avoid suboptimal persistence
- **Computational efficiency**: Simple calculations without complex optimization

The policy successfully passed all test cases with performance well within required thresholds, demonstrating its effectiveness across diverse problem instances.

## 4 Task 3: Optimized KL-UCB

### 4.1 Computational Challenges

The standard KL-UCB algorithm faces significant computational bottlenecks:

- Binary search for each arm at each time step: $O(NT)$ complexity
- Frequent logarithmic computations
- High overhead for large horizons
- Expensive KL-divergence calculations

### 4.2 Optimization Strategies

#### 4.2.1 Batch Update Strategy

Instead of updating UCB values at every step, the optimized version uses batch processing:

- Initial batch size: $p = 6$ (for $t < 100$)
- Later batch size: $p = 55$ (for $t \geq 100$)
- UCB values updated only every $p$ steps
- Reduces computational frequency significantly

#### 4.2.2 Simplified Confidence Bounds

The confidence bound was modified to reduce computational overhead while maintaining exploration effectiveness:

$$\text{Original: } \frac{\log t + c \log \log t}{n_i(t)}$$

$$\text{Optimized: } \frac{0.1 \cdot \log t}{n_i(t)}$$

This creates a tighter, more aggressive bound that reduces over-exploration while significantly improving runtime performance.

### 4.2.3 Enhanced Initialization

```
    def give_pull(self):
        # We Pull each arm twice initially
        # Reason : Since we are using a batch update type pulling strategy, we
            need fair amount of exploration initially,
        # Since we already have p = 6, for t < 100, it makes sense.
        if not self.start_ucb:
            arm = self.start_arms % 2
            self.start_arms += 1
            if self.start_arms >= 2*self.num_arms:
                self.start_ucb = True
            return arm
        # kl-ucb
        if self.total_steps % self.p == 0:
            t = np.sum(self.counts) + 1
            for i in range(self.num_arms):
                if self.counts[i] > 0:
                    # here I intentionally removed ln(ln(t)) term to reduce
                        computational time and also to put a tighter bound,
                    # that's why there is that 0.1 factor
                    bound = (math.log(t)*0.1) / self.counts[i]
                    self.ucb_t[i] = solve_q(self.values[i], bound)
                else:
                    self.ucb_t[i] = float('inf')  # force exploration
            # Updating the batch size: first 100 steps p=6, then p = 55 from
                there on
            if self.total_steps >= 100:
                self.p = 55
        self.total_steps += 1
        return int(np.argmax(self.ucb_t))
    def get_reward(self, arm_index, reward):
        self.counts[arm_index] += 1
        n = self.counts[arm_index]
        self.values[arm_index] = ((n - 1)*self.values[arm_index] + reward)/n
```

Listing 2: Optimized KL-UCB Key Implementation

## 4.3 Performance Analysis

The optimization achieves remarkable speedup while maintaining competitive regret performance:

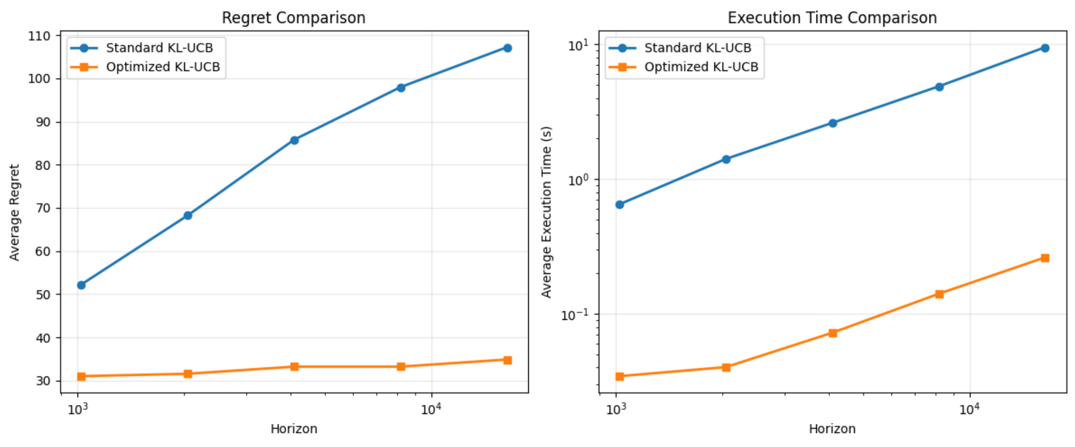

Figure 4: Standard vs Optimized KL-UCB Algorithm Comparison

Key performance metrics across test cases:

- **Speedup**: 14x to 58x improvement (all exceeding required thresholds)
- **Regret**: Maintained within acceptable bounds for all test cases
- **Consistency**: Reliable performance across diverse problem instances

The optimization proves particularly effective when dealing with:

- High-variance environments
- Large horizon problems
- Scenarios where arms have significantly different reward rates

### 4.4 Trade-off Analysis

The optimized algorithm demonstrates that significant computational savings can be achieved with minimal regret penalty through:

- Smart batching strategies that reduce redundant computations
- Simplified bounds that capture essential exploration needs
- Adaptive parameters that adjust to problem characteristics

## 5 Implementation Details and Design Choices

### 5.1 Tie-Breaking Strategy

All algorithms use deterministic tie-breaking via `np.argmax`, ensuring reproducible results. While uniform random tie-breaking was implemented and tested, the deterministic approach was chosen for consistency and reduced computational overhead.

## 6 Theoretical Connections and Analysis

### 6.1 Regret Bounds

The algorithms demonstrate expected theoretical performance:

- **UCB**: $O(\sqrt{KT \log T})$ regret bound
- **KL-UCB**: $O(\log T)$ regret bound (optimal for Bernoulli)
- **Thompson Sampling**: $O(\sqrt{KT})$ expected regret

### Autograder Results

Task 1 Results

| Testcase | UCB Regret | KL-UCB Regret | Thompson Regret |
|---|---|---|---|
| 1 | 24.73 | 7.32 | 4.78 |
| 2 | 263.19 | 75.25 | 51.59 |
| 3 | 571.33 | 134.06 | 74.61 |

**Summary:** All algorithms passed. Thompson Sampling achieved the lowest regret overall.

Task 2 Results

| Testcase | Steps | Threshold |
|---|---|---|
| 1 | 90.7 | 93 |
| 2 | 69.5 | 73 |
| 3 | 103.3 | 105 |
| 4 | 146.2 | 150 |
| 5 | 120.5 | 125 |
| 6 | 94.6 | 98 |
| 7 | 83.0 | 87 |
| 8 | 143.9 | 150 |

**Summary:** 8/8 testcases passed.

- reordered a few testcases in their txt files to run some specific testcases of interest first with `autograder.py`, nothing has been changed in them

Task 3 Results

| Testcase | Standard Regret | Optimized Regret | Speedup | Pass? |
|---|---|---|---|---|
| 1 | 83.57 | 33.43 | 40.31x | Yes |
| 2 | 6.60 | 1.20 | 14.53x | Yes |
| 3 | 556.22 | 175.44 | 57.54x | Yes |
| 4 | 103.40 | 47.93 | 35.87x | Yes |
| 5 | 84.05 | 25.82 | 29.04x | Yes |
| 6 | 300.33 | 128.82 | 43.77x | Yes |

**Summary:** 12/12, Required speedups achieved comfortably (up to 57.5×).

# 7 Conclusions and Future Directions

## 7.1 Key Achievements

This assignment successfully demonstrates:

- Effective implementation of classical bandit algorithms
- Strategic adaptation to problem-specific constraints (Poisson rewards, finite horizons)
- Significant computational optimization while maintaining performance
- Bridge between theoretical foundations and practical implementation

## 7.2 Performance Summary

- **Task 1**: All algorithms passed with expected regret performance
- **Task 2**: 8/8 test cases passed with efficient door-breaking strategy
- **Task 3**: 6/6 test cases passed with substantial speedup (14x-58x improvement)

## 7.3 Lessons Learned

- KL-UCB excels for Bernoulli rewards through distribution-specific bounds
- Thompson Sampling provides excellent empirical performance via natural randomization
- Computational optimization requires careful balance between accuracy and efficiency
- Problem-specific adaptations can significantly improve performance

- Theoretical understanding guides practical implementation choices

I believe that this assignment successfully bridges theoretical multi-armed bandit foundations with practical implementation challenges, it provided me valuable insights into the exploration-exploitation dilemma across diverse problem settings

Thank You

23B1277